Benchmarking Sim-2-Real Algorithms on Real-World Platforms

Benchmarking von Sim-2-Real Algorithmen auf Plattformen in der realen Welt

Bachelor-Thesis von Robin Menzenbach aus Gießen

Oktober 2019

TECHNISCHE UNIVERSITÄT DARMSTADT

Benchmarking Sim-2-Real Algorithms on Real-World Platforms
Benchmarking von Sim-2-Real Algorithmen auf Plattformen in der realen Welt

Vorgelegte Bachelor-Thesis von Robin Menzenbach aus Gießen

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr.Michael Gienger
3. Gutachten: M.Sc. Fabio Muratore

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Robin Menzenbach, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Datum / Date: Unterschrift / Signature:

_____ _____

Abstract

Learning from simulation is particularly useful, because it is typically cheaper and safer than learning on real-world systems. Nevertheless, the transfer of learned behavior from the simulation to the real word can impose difficulties because of the so-called 'reality gap'. There are multiple approaches trying to close the gap. Although many benchmarks of reinforcement learning algorithms exist, state-of-the-art sim-2-real methods are rarely compared. In this thesis, we compare two recent methods on Furuta pendulum swing up and ball balancing tasks. The performed benchmarks aim at assessing sim-2-sim and sim-2-real transferability. We show that the application of sim-2-real methods significantly improves the transferability of learned behavior.

Zusammenfassung

Das Lernen aus der Simulation ist besonders hilfreich, weil es günstiger und sicherer ist als das direkte Lernen auf den realen Systemen. Jedoch ist es nicht einfach, in der Simulation gelerntes Verhalten direkt in die Realität zu übertragen. Grund dafür ist der sogenannte 'reality gap'. Es gibt mehrere Ansätze diese Lücke zwischen Relität und Simulation zu schließen. Obwohl einige Benchmarks auf Reinforcement Learning Methoden durchgeführt wurden, gibt es nur wenige Auswertungen von sim-2-real Methoden. In dieser Thesis werden wir zwei Methoden evaluieren. Die zu lösenden Aufgaben werden das Hochschwingen und Balancieren eines Furuta Pendels und eines Balls sein, der auf einer Platte balanciert werden muss. Wir zeigen, dass die Anwendung von sim-2-real Methoden die Übertragbarkeit gelernten Verhaltens signifikant verbessert.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
ADR	active domain randomization
ARPL	adversarially robust policy learning
BO	Bayesian optimization
DR	domain randomization
FGSM	fast gradient sign method
GAE	generalized advantage estimation
MDP	Markov secision process
PPO	proximal policy optimization
RL	reinforcement learning
SVPG	Stein variational policy gradient
TRPO	trust region policy optimization
UDR	uniform domain randomization

List of Symbols

Notation	Description
a	an action in the MDP (can be a vector)
$\mathscr{A}$	action space
μ	agent particle
φ	probability of applying a perturbation
γ	discount factor
D	discriminator
E	simulation environment
ξ_{nom}	nominal settings for the physics parameters
N_{rand}	number of randomized parameters
ω	observation noise
θ	parameters of a policy
$\boldsymbol{\theta}$	vector of parameters from a probability distribution
ϕ	parameters for an agent particle
ξ	physics configuration

π	agent policy
η	a loss function on the policy
ν	scalar factor for performing policy gradient steps
$\varkappa$	process noise
Ξ	randomization space
τ	rollout
S	simulator
s	a state belonging to the MDP (can be a vector)
$\mathscr{S}$	state space
s_t	the state in timestep t
$\mathscr{P}$	transition function

List of Operators

Notation	Description	Operator
Q^π	state-action-value function	$Q^\pi(s, a)$
$\hat{}$	an estimate	$\hat{\bullet}$
$\mathbb{E}$	expectation	$\mathbb{E}(\bullet)$
D_{KL}	Kullback-Leibler divergence	$D_{KL}(\bullet \| \bullet)$
log	the natural logarithm	$\log(\bullet)$
$\mathscr{N}$	a normal distribution parameterized with mean and standard deviation	$\mathscr{N}(\mu, \sigma)$
R	discounted return of a trajectory	$R(\tau)$
r	reward function	$r(s, a)$
$\mathscr{U}$	a uniform distribution parameterized with a center and a halfspan	$\mathscr{U}(\mu, \sigma)$
V^π	state-value function	$V^\pi(s)$

1 Introduction and Motivation

The rise of reinforcement learning (RL) allowed to solve diverse tasks in robotics and other areas [1, 2, 3]. Because many of them require behavior which would be difficult to describe analytically, learning from experience is a promising way to succeed. Yet, the majority of algorithms needs a lot of data for training. However, the sampling of this data can be highly inefficient or expensive on the real platforms. This is why training in simulation is often inevitable in deep RL. Naturally, a simulation cannot perfectly replicate the real world but uses some model to resemble the real physics. One could say that simulators are inherently inaccurate as they do not model all physical effects. Closing this so-called 'reality gap' [4] is a vivid research area.

Generally, sim-2-real approaches aim to gain knowledge in simulation which can be transferred to the target domain e.g. the real world. As an example, system identification can be used to improve the physics parameters of the simulation to match the target domain. Another promising approach is the variation of the source domain (the simulator) while training. A perturbation of the physics parameters allows the policy to be more robust and versatile. For domain randomization, the parameters are sampled from a distribution. In the simplest form fixed distributions are used to sample physics parameters for training. Further, real data can be used to improve the parameters of the distribution, as in[5]. Notably, the distribution is adapted instead of the parameters themselves. Ideally, learned policies would succeed in the real world without requiring fine-tuning because to the policy it would appear to be just another perturbation [6]. Nevertheless there are even more advanced ways to sample the source environments. Some train additional agents to sample physics parameters [7] whilst others adversarially apply perturbations to lower the performance of the trained policies [8]. Many successes have been made in closing the gap between simulation and reality [9, 10]. In this manner, the search over the source domains becomes a RL problem. Benchmarking different algorithms on different platforms gives more insights on their performance. Hence different algorithms are benchmarked on different real world platforms.

Learning tasks in simulation is very promising. There is potential in sim-2-real methods because the learned behavior may be applicable to the real world. Such methods are an important research topic. However there are only very few benchmarks comparing new methods. When new methods are proposed they are often compared to methods like system identification and rarely to other state-of-the-art methods. Therefore, we want to compare two state-of-the-art methods by performing sim-2-real and sim-2-sim experiments, explained in chapter 5.

2 Related Work

Recently, there have been major successes in deep RL [11, 12, 1]. Training in simulation is inevitable because it is cheaper and safer than training on the real systems. Hence there is a great interest in transferring learned behavior to the real word. In recent years, a lot of research was conducted on finding ways to transfer behavior from simulation to reality (sim-2-real transfer). There are many works on closing the so-called reality gap [4, 13, 14, 15, 16, 6, 17]. Two ways to improve the transferability of learned behavior are (i) system identification [18] (ii) domain randomization, which has also been applied in general deep learning research [19]. There are also many works on benchmarking RL algorithms [20, 21, 22, 23]. But there are not many benchmarks and comparisons of recent sim-2-real methods applied to the same RL methods [24]. In this work we want to benchmark two algorithms and test the transferability of behaviors they learn.

3 Foundations

3.1 Reinforcement Learning

In order to understand how different algorithms aim at solving the examined tasks, we need to introduce RL. In general, it can be described as learning through interaction. The agent interacts with the environment and observes the outcome. Additionally, the agent receives a scalar reward signal, which quantifies if the decision was beneficial regarding the task's goal. Thus, the agent can adapt its behavior accordingly to receive more reward in the future. Many theories in RL originated in psychology and behavioral science [25]. One could say that RL algorithms learn by trial and error. Moreover, it is important to find the right balance between exploration and exploitation.

3.1.1 Markov secision process

RL is often described using a Markov secision process (MDP) which is the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. In general RL is a representation of a planning problem. Where $\mathcal{S}$ is the state space, $\mathcal{A}$ the actions space, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_+$ is the probability of a state transition from one state to another when choosing a certain action and $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward that corresponds to the transition. In each time step, the agent perceives the environment and performs an action which causes a transition to the next state as well as the reception of a reward signal (see Figure 3.1a and Figure 3.1b). The policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$ is a probability distribution over the action given a state. It is parameterized by a set of parameters θ. Sequences of steps and actions generated by taking the actions proposed by the policy are called trajectories or rollouts. A rollout $\tau = (s_0, a_0 \ldots, s_T)$ which was sampled by a policy $a_t \sim \pi(a_t|s_t)$, also has an associated return.

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t).$$

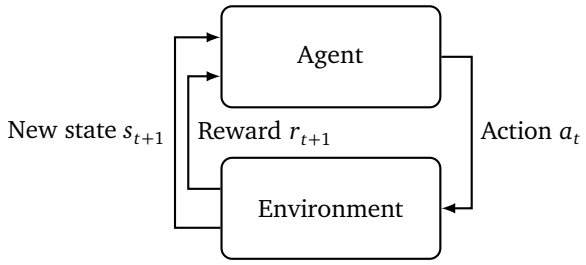
The expected discounted return of a policy is

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)], \tag{3.1}$$

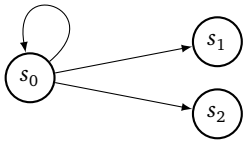
where γ is the discount factor which balances the value of future against immediate reward. If $\gamma = 0$, the agent is only concerned about immediate rewards while a value close to $\gamma = 1$ yields a farsighted agent. Thus the goal of RL can be formulated as

$$\theta^* = \underset{\theta}{\operatorname{argmax}} J(\theta).$$

Where π_{θ^*} is the optimal policy that maximizes the expected return.



(a) In a MDP the agent perceives the environment and interacts by selecting an action. Then the environment transitions to a new state resulting in a reward.



(b) This exemplary MDP consists of three states with three actions.

Figure 3.1.: Markov secision process

3.1.2 reinforcement learning algorithms

Most RL algorithms can be divided in two classes: (i) value function based methods and (ii) policy search algorithms. This chapter contains a brief introduction to the landscape of available algorithms while focusing on the ones used in the sim-2-real benchmark. More comprehensive surveys on different algorithms can be found in [26, 27, 28].

Methods based on value functions

The methods based on value functions estimate the expected return at any given state through a state-value function

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi_\theta, s_0=s}[R(\tau)].$$

The expected return of a policy at a given state and an initial action is approximated by the state-action-value function

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi_\theta, s_0=s, a_0=a}[R(\tau)].$$

Value function based methods use a policy that greedily takes the action with the highest predicted value.
In the easiest setting, a table with the values is maintained for each action and state. Those values are updated by

$$v_{k+1}(s) = \sum_{a\in\mathscr{A}} \pi(a|s) \sum_{s'\in\mathscr{S}} \mathscr{P}(s,a,s')\big(r(s,a) + \gamma v_k(s')\big).$$

The greedy policy is given as

$$\pi'(s) = \operatorname*{argmax}_{a\in\mathscr{A}} q(s,a).$$

If a table is used to maintain the different values, the state and action spaces need to be discretized. However there are also methods that approximate the value function using neural networks [29]. This allows a continuous state space, while the action space still needs to be discritized. Since we will be testing the algorithms in continuous control tasks, a complete introduction to value based methods can be found in [25].

Methods based on policy search

Policy search methods directly optimize the policy without a value function. Instead, the parameters of a policy π_θ are optimized directly. Although policy gradient methods are dominant, gradient-free approaches exist [30].

Policy Gradient Methods
In order to obtain an estimate of the policy gradient, one can use deterministic approximations of the trajectories generated by the policy. Deterministic approximations are only possible if a model of the system's dynamics is available. In model-free settings an estimate of the expected return is used instead. Since stochastic rollouts can not be differentiated, an approximation of the gradient is necessary.

The REINFORCE algorithm [31] uses the likelihood-ratio gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log p(\tau|\theta) R(\tau)\right].$$

Because these estimates rely on empirical data, the variance is high. By subtracting a baseline, it is possible to reduce the variance of the gradient estimate. Among many possible baselines, the average return over multiple rollouts is the simplest one.

Actor-critic methods
Actor-critic methods combine concepts of policy gradient methods with value function based ones. They use a parameterized policy (actor) that is trained using feedback from a value function (critic). This value function is inferred from empirical data and provides a baseline for the policy gradient estimation. Advantage actor-critic methods estimate the gradient based on advantage instead of reward. The advantage is defined as $A^{\pi}(s_t,a_t) := Q^{\pi}(s_t,a_t) - V^{\pi}(s_t)$.
The policy parameters are updated using a stochastic gradient ascent algorithm and the gradient estimate

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right],$$

where Ψ_t can be the baselined reward $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$ or the advantage. Moreover, generalized advantage estimation (GAE), proposed in [32], can be used to approximate the policy gradient based on trajectory data.

Proximal policy optimization

We will use proximal policy optimization (PPO) [12] to train our policies. We also could use other algorithms, but PPO showed impressive results while being simpler. Moreover, some of the benefits of trust region policy optimization (TRPO) also apply to PPO although it is easier to implement. One of them is a mechanism that bounds the policy updates. This is beneficial because the advantage estimate gets less accurate when the new policy moves away from the old one. In PPO, the gradient is estimated as

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]. \tag{3.2}$$

While the advantage is estimated using GAE. TRPO, as proposed in [2], formulates a constrained optimization problem:

$$\begin{aligned} \underset{\theta}{\text{maximize}}\ & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \\ \text{s.t. } & \hat{\mathbb{E}}_t \left[\mathrm{KL}[\pi_{\theta_{old}}(a_t|s_t), \pi_\theta(a_t|s_t)] \right] \leq \delta \end{aligned}$$

Instead of maximizing this constrained problem, PPO maximizes a surrogate objective:

$$\begin{aligned} L(\theta) &= \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\, \hat{A}_t) \right] \\ r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \end{aligned} \tag{3.3}$$

where $r_t(\theta)$ is the likelihood-ratio. The optimization of the likelihood increases the impact of actions which were less probable with the previous parameters θ_{old}. However, the presented method is only one possible variant of PPO.
Since PPO methods represent an actor-critic architecture, they also include a value function that has to be updated. Commonly, a variance-reduced advantage-function is estimated by a state-value function which is learned using a neural network. After computing the estimated advantages, the policy gradient estimate from (3.2) is evaluated. The policy's parameters are optimized using Adam [33]. As shown in Algorithm 1, PPO uses multiple epochs of minibatches to update the policy.

for *iteration* $= 1, 2, \ldots$ **do**
  for *actor* $= 1, 2, \ldots, N$ **do**
    Sample from environment using $\pi_{\theta_{old}}$ for T timesteps
    Estimate advantages $\hat{A}_1, \ldots, \hat{A}_T$
  Optimize surrogate loss (3.3), using K epochs and minibatches
  $\theta_{old} \leftarrow \theta$
  Fit the value function by regression on the mean squared error via some gradient descent algorithm.

Algorithm 1: Core part of PPO in an actor-critic architecture, as proposed in [12]

3.2 Stein Variational Policy Gradient

In RL, it is necessary to trade off exploitation against exploration. Stein variational policy gradient (SVPG) provides a framework which uses multiple particles to explicitly encourage parameter exploration. A particle is an RL agent (e.g. an actor-critic architecture). Since optimizing the parameters of a neural-network is a non-convex optimization problem, trained policies are often sensitive to their initializations. The SVPG method proposed in [34] is built on the Stein variational gradient descent method [35] which simultaneously explores and exploits in multiple policies (as particles). The utility function for a given policy (3.1) should be maximized. But instead of searching one parameter set θ, SVPG seeks a distribution over parameters $q(\theta)$. A default distribution q_0 is used to incorporate prior domain knowledge. Hence the problem can be formulated as

$$\max_q \left\{ \mathbb{E}_{q(\theta)}[J(\theta)] - \alpha D_{KL}(q||q_0) \right\},$$

where $D_{KL}(q||q_0) = \mathbb{E}_{q(\theta)}[\log q(\theta) - \log q_0(\theta)]$. As shown in [34], the KL divergence can be simplified to be the entropy of q, by choosing a constant prior. This is why an exploration in the parameter space according to maximum entropy takes place. The authors also claim, that this formulation is equivalent to a Bayesian formulation

$$\underbrace{q(\theta)}_{\text{posterior}} \propto \underbrace{\exp\left(\frac{1}{\alpha} J(\theta)\right)}_{\text{likelihood}} \underbrace{q_0(\theta)}_{\text{prior}}. \tag{3.4}$$

Moreover, Liu et al. [35] show, that a closed form solution for an update of multiple particles θ_i exists. An update decreases the KL divergence between the particles and the target distribution

$$\theta_{i+1} \leftarrow \theta_i + \epsilon\phi(\theta_i),$$
$$\phi^*(\theta) = \mathbb{E}_{\vartheta\ \rho}[\nabla \log q(\vartheta)k(\vartheta,\theta) + \nabla_\vartheta k(\vartheta,\theta)],$$

where $k(\cdot,\cdot)$ is a kernel function, ϵ a scalar factor and ρ the probability distribution over θ. Using the empirical mean over the particles instead of the expectation yields the Stein variational gradient

$$\hat{\phi}(\theta_i) = \frac{1}{n}\sum_{j=1}^{n}\left[\nabla_{\theta_j}\log q(\theta_j)k(\theta_j,\theta_i) + \nabla_{\theta_j}k(\theta_j,\theta_i)\right].$$

By applying (3.4), the update step of SVPG becomes

$$\hat{\phi}(\theta_i) = \frac{1}{n}\sum_{j=1}^{n}\left[\nabla_{\theta_j}\left(\frac{1}{\alpha}J(\theta_j)\log q_0(\theta_j)\right)k(\theta_j,\theta_i) + \nabla_{\theta_j}k(\theta_j,\theta_i)\right],$$

where the parameter α is responsible for trading-off exploration against exploitation. A large value for α would drive the particles towards the prior, while a very small value would reduce the algorithm to n independent policy gradient algorithms. The last term of the sum is responsible for repulsion between the particles.

input : Learning rate ϵ, kernel $k(x', x)$, temperature α, initial particles θ_i
for *iteration* $t = 0, 1, \ldots, T$ **do**
  for *particle* $i = 0, 1, \ldots, n$ **do**
    Rollout particle i
    Estimate $\nabla_{\theta_i} J(\theta_i)$
  for *particle* $i = 0, 1, \ldots, n$ **do**
    $\Delta\theta_i \leftarrow \frac{1}{n}\sum_{j=1}^{n}[\nabla_{\theta_j}\left(\frac{1}{\alpha}J(\theta_j)\log q_0(\theta_j)\right)k(\theta_j,\theta_i) + \nabla_{\theta_j}k(\theta_j,\theta_i)]$
    $\theta_i \leftarrow \theta_i + \epsilon\Delta\theta_i$

Algorithm 2: Core part of Stein Variational Policy Gradient as proposed in [34].

3.3 Reality Gap

RL has shown impressive results in (simulated) robotic control tasks. Using simulators as a tool to learn control tasks with RL methods significantly reduces wear and risk of damage on the hardware. Moreover, it is cheaper and faster to acquire data in simulation. But simulators can only approximate the real world and therefore have an innate modeling error. Therefore, a transfer to the real system is often not directly possible [4]. One way to deal with the 'reality gap', is to improve the simulator in terms of accuracy. This can e.g. be done by system identification on the real system and transferring the learned parameters to the simulator. But unfortunately the parameters can be dependent on environmental factors such as the temperature. Moreover, system identification does not change which physical effects are modeled by the simulator. Hence, it is likely that there is still a discrepancy between the real world and the simulator after system identification. The learner can also exploit flaws in the simulator which can lead to overfitting on the simulation. Another approach to improve the transferability is to simulate observation noise or action delays. One could also learn a policy in simulation and then improve it on the real system. But this is only feasible if the system cannot be damaged easily. Domain randomization (DR), introduced in section 3.4, tries to tackle the 'reality gap' by randomizing the domain parameters during training. The idea behind DR is that the real world to the agent would look as if it was just an instance of the simulation.

3.4 Domain Randomization

DR is a way to approach the sim-2-real transfer and works by perturbing the physics parameters in the source domain which is simulated. The idea behind DR is that the agent perceives the reality as another instance of the simulation. Ideally, learned policies can then be transferable to the target domain without any need of fine tuning. When performing DR, it is important to define a set of parameters to be randomized $rand = \left\{\xi^{(i)}\right\}_{i=1}^{N_{rand}}$. Further, bounding each paramter on a closed interval is useful to prevent the simulation from using implausible physics, e.g. negative masses, $\left\{\left[\xi_{low}^{(i)}, \xi_{high}^{(i)}\right]\right\} \forall i \in \{i \in \mathbb{N} \mid 1 \leq i \leq N_{rand}\}$. Different parameter setups ξ are sampled from a randomization space $\Xi \subset \mathbb{R}^{N_{rand}}$. These setups are passed to a simulator S in order to create the physics environments E_ξ. DR aims to model discrepancies between the source and target domain as variability in the source domain[17]. Generally, DR is able to perturb many aspects of the MDP while γ and r have to stay the same. Hence, DR generates a set of MDPs which are similar but nevertheless can impose different challenges on the agent policy π.

3.4.1 Uniform Domain Randomization

When performing UDR, each domain parameter is uniformly sampled from the randomization space Ξ. $\xi_i \in [\xi_i^{low}, \xi_i^{high}], i = 1, \ldots, N_{rand}$. Therefore, the agent experiences a variety of domain settings during training. In [6] as an example of UDR, the parameters are sampled before every rollout and kept until finished. After sampling rollouts τ_i from a randomized environment E_i, the policy is updated using a policy gradient method.

input : randomisation space Ξ, simulator S
initialize : agent policy π_θ
foreach *episode* **do**
  for $i = 1$ **to** N_{rand} **do**
    $\xi^{(i)} \sim U\left[\xi_{low}^{(i)} \xi_{high}^{(i)}\right]$
  $E_i \leftarrow S(\xi_i)$ // Create a randomized environment with parameters ξ
  rollout $\tau_i \sim \pi$ on E_i
  $\mathscr{T}_{rand} \leftarrow \mathscr{T}_{rand} \cup \tau_i$ //append rollout to buffer
  foreach *gradient step* **do**
    // update policy using the samples from all randomized environments $\mathscr{T}_{rand}$
    $\theta \leftarrow \theta + \nu \nabla_\theta J(\pi_\theta)$

Algorithm 3: Uniform Domain Randomization

3.4.2 Domain Randomization with Advanced Randomization Spaces

The approach of sampling the source domains from a uniform distribution relies on the assumption that the broad space which is explored contains environments which are valuable for training the agent. However, some approaches are targeted at an improvement of the sampling space by using received returns as well as trajectories from simulation or target domain. There are approaches with the incentive to minimize the discrepancy between the trajectories sampled from source and target domains respectively [36].

The algorithms which we chose to evaluate are both using additional utility functions which they aim to maximize. In active domain randomization (ADR), this is done by formulating another RL problem around the policy gradient method. The second one uses adversarial samples to generate physically plausible noise which aims to yield more robust policies in the training process with underlying policy gradient methods. Since both methods can be wrapped around a subroutine policy gradient method, we will use PPO as the underlying algorithm for both.

3.4.3 Active Domain Randomization

The method of ADR [7] formulates the search of a suitable randomization space Ξ as an RL problem itself. Therefore, the randomization space Ξ is treated as a search space. The goal is to find configurations ξ which provide the most improvements when trajectories sampled in the environment $E = S(\xi)$ are used for training.

The authors compare this procedure to Bayesian optimization (BO), where an acquisition function determines where to evaluate next. The acquisition function finds the right balance between exploration and exploitation. However, BO is not well suited in searching optimal ξ in Ξ because the objective function is non-stationary, because the policy π is updated in every iteration. Nevertheless, different objective functions can render BO suitable.

The proposed way to circumvent this limitation is to actively search the space for training environments given the most recent policy. The method formulates the analog to the aquisition function as an RL problem itself. Several agents μ_ϕ i.e. particles are trained to propose randomization spaces. The states they recive are configurations ξ and the actions are changes which will be applied to them. Moreover, a discriminator is used with the intention to provide a reward signal to the particle policies. The discriminator is trained to predict if a trajectory τ_i was sampled from a reference environment E_{ref} or from a randomized environment which has been proposed by one of the agents μ_ϕ.

$$R_{discriminator} = \log D(y|\tau_i \sim \pi(\theta; E_i)), \tag{3.5}$$

where y is a binary variable which indicates if the trajectory was sampled in the reference environment and D is the discriminator which outputs the probability that a sample has been recorded in a randomized environment. This probability is used as a reward signal to train the domain randomization agents. Accordingly, the agents which generated environments with trajectories easily recognizable as non reference ones, are rewarded.
As an intuition, the authors claim that trajectories that can be easily identified as not from the reference environment, provide more information to the learner. The agents are jointly trained using SVPG (see section 3.2). Since the generated configuration of all agents are used to sample trajectories which are then used to update the policy, a high variety of physics configurations used for training is still given. In contrast to simpler methods like UDR, ADR uses the reward generated by the discriminator to find environments that are more challenging for the agent. After the rollout of the policy π in the environments generated by each of the particles, the trajectories are used to perform multiple policy gradient steps. Finally, after the samples from every proposed configuration are collected, the trajectories are used to train the discriminator using ADAM [33]. ADR can be found in Algorithm 4.

input : randomisation space Ξ, simulator S, reference parameters ξ_{ref}
initialize : agent policy π_θ, SVPG agent particles μ_ϕ, discriminator D, reference environment $E_{ref} \leftarrow S(\xi_{ref})$
foreach *episode* **do**
  foreach *particle* **do**
    rollout $\xi_i \sim \mu_\phi$
  foreach ξ_i **do**
    $E_i \leftarrow S(\xi_i)$
    rollout $\tau_i \sim \pi_\theta(E_i)$ and $\tau_{ref} \sim \pi_\theta(E_{ref})$
    $\mathcal{T}_{rand} \leftarrow \mathcal{T}_{rand} \cup \tau_i$
    $\mathcal{T}_{ref} \leftarrow \mathcal{T}_{ref} \cup \tau_{ref}$
    foreach *gradient step* **do**
      $\theta \leftarrow \theta + \nu \nabla_\theta J(\pi_\theta)$ //using $\mathcal{T}_{rand}$
  foreach $\tau_i \in \mathcal{T}_{rand}$ **do**
    Calculate the reward using the discriminator D, see (3.5)
  foreach *particle* **do**
    Update particle μ_ϕ using SVPG
  foreach *discriminator gradient step* **do**
    Update D using $\mathcal{T}_{rand}$ $\mathcal{T}_{ref}$

Algorithm 4: Active domain randomization as proposed in [7]

3.4.4 Adversarially Robust Policy Learning

Adversarially robust policy learning (ARPL) [8], aims at inducing physically plausible perturbations to different aspects of simulations. Adversarial perturbations are chosen actively in the course of training. These adversarial examples are generated using the fast gradient sign method (FGSM) which has been introduced in [37]:

$$\delta = \varepsilon \operatorname{sign}(\nabla_s \eta(\pi_\theta(s))).$$

A loss function is used to create adversarial examples that increase the norms of the policy's output:

$$\eta(a) = ||a||_2^2.$$

The agent thereby increasingly experiences dynamics and observations which result in higher actions. The idea is that higher actions lead to instability.

There are three locations in the model where ARPL can apply adversarial examples ξ, $\varkappa$ and ω. The dynamical system is described as:

$$\begin{aligned} s_{t+1} &= f(s_t, a_t, \xi) + \varkappa \qquad \text{and} \\ o_t &= g(s_t) + \omega, \end{aligned}$$

where f is the state transition function and g the observation function.
In the model for transitioning to the next state, there are two points where adversarial samples can be applied. These are the dynamics parameters ξ and the process noise $\varkappa$.
Additionally, the observation model represents the relationship between the state and what the agent perceives. Observation noise ω can be applied here.
First, in case of the observation noise ω, the adversarial sample is calculated and then added to the observation with a scaling factor. Second, the process noise is applied by adding the scaled adversarial to the systems state. Lastly, in order to get a gradient of the loss function with respect to the dynamics parameters, the state is augmented $\bar{s} = [s, \xi]$. But for the adversarial only the component of the physics configuration is used: $\nabla_{\bar{s}} = \left[0, \nabla_{\xi}\right]$. The application of the dynamics noise is performed by sampling a configuration from a uniform distribution $\mathcal{U}(0.5 \cdot \xi_{nom}, 1.5 \cdot \xi_{nom})$ at the beginning of each episode and then adding adversarial samples at each time step.
In contrast to ADR and UDR, the adversarial samples are applied at every time step. But a Bernoulli distribution is sampled to determine if the adversarial is applied or not.

input : randomisation space Ξ, simulator S, number of rollouts k, probability of applying perturbation φ, ϵ scalar factor for the adversarial
initialize : agent policy π_θ, environment E
foreach *episode* **do**
  for *rollout* $i = 1, \ldots, k$ **do**
    for $t = 1, \ldots, T_k$ **do**
      Sample s_t, a_t from the environment E using π_θ
      Compute adversarial perturbation δ using FGSM
      apply perturbation δ with probability $Bernoulli(\varphi)$ and factor ϵ
  $\theta_{i+1} \leftarrow \theta_i + \nu \nabla_\theta J(\pi_\theta)$ // update the policies using the rollouts $\tau_0 ... \tau_k$

Algorithm 5: Core part of adversarially robust policy learning as proposed in [8]

Ability to be parallelized

For RL methods or their simulation environments, the ability to be parallellized is important since the amount of steps sampled from a simulator can become very large. If trained using DR, the amount of required samples is usually even larger [7]. An important feature that (plain) PPO, ADR, ARPL and UDR share is the ability to parallelize the simulation environments for sampling trajectories.

4 Method

Since there is an abundance of new sim-2-real methods, it is infeasible to compare large quantities of approaches. The goal of our experiments is the benchmark of two state-of-the-art algorithms in sim-to-sim and sim-2-real settings. Therefore the platforms will be consistently used to investigate if successes in the sim-2-sim transferability can be deployed on the real systems. Further, the philosophy of our assessments is to value the final performance of any method greater than the process of training. The principal task is to achieve transferability of agent policies to the real system.

4.1 Platforms

To evaluate the transferability of trained policies, it is crucial to have platforms which are available in both, the real world and simulation.

4.1.1 Ball Balancer

The ball balancer consists of a plate which can be tilted with two motors without a prebuilt controller but directly receiving a motor voltage. They move cylinders up and down on which the plate rests. The basic task is to move a ball to a specific

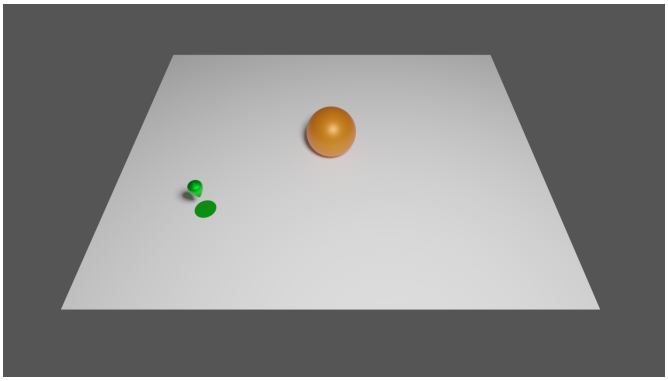


Figure 4.1.: The general task for the ball balancer is to move the ball to a goal point (illustrated in green). The plane can be tilted along both axes.

goal point by tilting the plate (see Figure 4.1). Naturally, this task is nonlinear and of high interest, since balancing is an important objective in robotics [38]. The real world counterpart that has been used is manufactured by Quanser Inc. It uses a camera to sense the position of the ball as well as encoders to observe the position of the base cylinders. The control signal consists of two scalar values corresponding to the voltages of the motors.
When used for domain randomization, the parameters which can be changed in simulation are presented in Table A.1 with their nominal values.

The states are also the observations which are fed into the control policies consist of

$$s = \begin{bmatrix} \theta_1 & \theta_2 & x_{ball} & y_{ball} & \dot{\theta}_1 & \dot{\theta}_2 & \dot{x_{ball}} & \dot{y_{ball}} \end{bmatrix}.$$

Here, θ_1 and θ_2 are the joint angles measured by encoders mounted on the gears that are also attached to the poles on which the plate rests, see Figure 4.2. The measurements x_{ball} and y_{ball} are the coordinates of the ball, estimated using a camera. The corresponding velocities are also part of the observation.
The reward function we use in this task, is based on the quadratic error.

$$\begin{aligned} R(s,a) &= \exp(-c * (e_s \cdot (\mathbf{Q} \cdot e_s) + e_a \cdot (\mathbf{R} \cdot e_a))), \\ e_s &= s_{des} - s \text{ and} \\ e_a &= a \end{aligned} \tag{4.1}$$

where $c = -\log(0.0001)/c_{worst}$ and c_{worst} is the wost case cost. The error vector e_s represents the difference in each dimension. Further, the action itself is used as e_a to punish high actions.

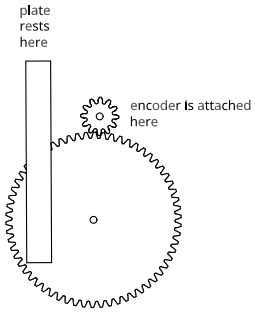


Figure 4.2.: Each of the ball balancer's motor units has a gearbox connected to its motor. The last gear is connected to a pole which acts as a rest for the plate. This gear is also connected to an encoder.

Balancing a ball on the ball balancer

On the ball balancer, the goal is to maneuver a ping-pong ball from a starting position to a desired position by tilting the plate. The reward is determined by a reward function (4.2). The weights for the error term are:

$$\mathbf{Q} = \text{diag}(1, 1, 1000, 1000, 0.01, 0.01, 0.5, 0.5) \qquad \text{and}$$
$$\mathbf{R} = \text{diag}(0.01, 0.01).$$

4.1.2 Furuta Pendulum

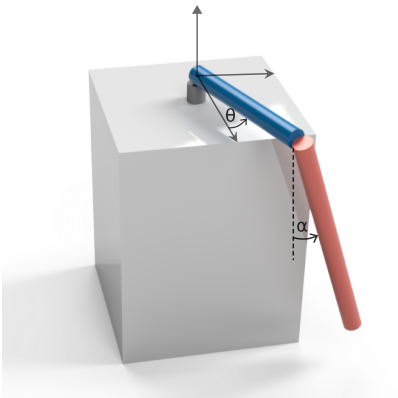


Figure 4.3.: A schematic of the Furuta pendulum platform.

The Furuta pendulum could be compared to a cartpole in circular shape. It was invented in 1992 by Katsuhisa Furuta [39]. Ever since, it has been a typical example for a nonlinear oscillator in system control engineering. As shown in Figure 4.3, it consists of two poles attached to each other perpendicularly. The tip of the first pole is attached to gear which can be driven by a motor. The motor's voltage is the only controllable value, therefore the system is underactuated. As for the ball balancer, the real world counterpart has been manufactured by Quanser Inc. The state and observations space consists of

$$s = \begin{bmatrix} \theta & \alpha & \dot{\theta} & \dot{\alpha} \end{bmatrix}^{\mathrm{T}}$$
$$obs = \begin{bmatrix} \sin\theta & \cos\theta & \sin\alpha & \cos\alpha & \dot{\theta} & \dot{\alpha} \end{bmatrix}.$$

The primary task on this platform is the nonlinear control task of reaching a desired point s_{des}. The reward function on this task can be formulated as

$$r(s,a) = \exp(-c(e_s \cdot (\mathbf{Q} \cdot e_s) + e_a \cdot (R \cdot e_a))), \tag{4.2}$$
$$e_s = s_{des} - s \text{ and}$$
$$e_a = a$$

with e_s being the difference between desired and actual state and e_a being the applied action. The equations of motion of the Quanser Qube are given by

$$\left(m_p L_r^2 + \frac{1}{4} m_p L_p^2 - \frac{1}{4} m_p L_p^2 \cos^2\alpha + J_r\right)\ddot{\theta} + \left(\frac{1}{2} m_p L_p L_r \cos\alpha\right)\ddot{\alpha} + \left(\frac{1}{2} m_p L_p^2 \sin\alpha\cos\alpha\right)\dot{\theta}\dot{\alpha}$$
$$-\left(\frac{1}{2} m_p L_p L_r \sin\alpha\right)\dot{\alpha}^2 + D_r\dot{\theta} = \frac{k_m(u - k_m\dot{\theta})}{R_m},$$
$$\left(\frac{1}{2} m_p L_p L_r \cos\alpha\right)\ddot{\theta} + \left(J_p + \frac{1}{4} m_p L_p^2\right)\ddot{\alpha} + D_p\dot{\alpha} - \left(\frac{1}{4} m_p L_p^2 \cos\alpha\sin\alpha\right)\dot{\theta}^2 + \frac{1}{2} m_p L_p g \sin\alpha = 0.$$

The nominal parameters are given in Table A.2.

Swing up on a Furuta pendulum

The task of the Furuta pendulum is to stabilize the pole in the upright position $s = \begin{bmatrix} 0 & \pi & 0 & 0 \end{bmatrix}^T$ the reward the agent receives is determined by the reward function in (4.1) with

$$\mathbf{Q} = \text{diag}(0.2, 1, 0.02, 0.005),$$
$$R = 0.003.$$

These weights on the reward function results in a high reward when the pole is upright. The error in the other observable dimensions has a minor influence to the overall reward. We define a rollout as successful, if the following statement holds true for the last second of execution

$$\text{abs}\left(obs_t - \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 0 \end{bmatrix}^T\right) < \begin{bmatrix} 0.2 & 0.2 & 0.02 & 0.02 & 0.1 & 0.1 \end{bmatrix}^T, \tag{4.3}$$

where abs is the element wise absolute value.

4.2 Training

The training process differs among the different algorithm-platform pairs. Since all methods use PPO to update the policy, the main difference lies in the randomization of the physics parameters i.e. the creation of configurations ξ.

The baseline method of plain PPO is used without any domain randomization. All other methods build upon this baseline by providing configurations $\xi \in \Xi$ for the simulator. Then, the provided configurations are used to sample trajectories which are then used to perform policy gradient steps using PPO. The randomization space Ξ is bounded relative to the nominal parameters ξ_{nom} of the particular platforms. Specifically, the bounds are set to $\xi_{low} = 0.5 \cdot \xi_{nom}$ and $\xi_{high} = 1.5 \cdot \xi_{nom}$. It is important to note that these bounds are not enforced in evaluation of trained policies. The corresponding nominal parameters can be found in Table A.1 and Table A.2.

During training, the algorithms that were introduced in section 3.4 are executed. Yet, due to the chance of late divergence, the new policy is saved after each policy update. In consonance with our increased interest to find the best possible performance, we use the best policy or – in case of ambiguity – a set of the best policies. Moreover, the duration of training is actually not important. Neither is the number of required simulator steps, because these are proportional to computation time. Therefore, it would be necessary to measure the overall computation time. But since the the final performance is the most important aspect we refrain from probing any computation time at all. For the sake of ruling out the chance of coincidentally discovering good policies, the algorithms are trained on diversified random seeds.

4.3 Evaluation

According to the goal of benchmarking several trained policies using a variety of domain randomization methods, we will perform two kinds of assessments. First, sim-2-sim experiments can be used to see if transferability to another configuration ξ is possible. Second, sim-2-real experiments show the policy's transferability to the real world application. Although the sim-2-real experiments investigate the fundamental question of transferability way better, sim-2-sim samples can be used to foster understanding about the different capabilities of the algorithms. Moreover, sim-2-real transfer could fail entirely which would render a comparison of some methods impossible if there is no data from sim-2-sim experiments.

In assessments on simulated environments, different criteria will be evaluated. First, the performance of each policy in a simulated environment with the nominal physics $E = S(\xi_{nom})$ is assessed. It is important to note, that the nominal environments can differ considerably from the ones used in training. Therefore, a good performance on the nominal parameters is not guaranteed.

DR is applied with the goal to improve the agent's performance in multiple environments. Hence it is a good metric to evaluate the different policies in physical configurations that differ from the nominal ones. Since the dimension of the randomization space Ξ is too high to be visualized entirely, we use two other approaches to visualize the ability to perform in perturbed environments.

On the one hand, the physics parameters will be sampled from a random distribution at each rollout. Subsequently, the return and success rate is computed.
On the other hand, only a subset of the randomization space will be used to generate configurations ξ in a grid. In order to be able to plot this grid, two dimensions will be used. For each physics parameter which is perturbed, an interval and a number of steps is necessary. Furthermore, the interval is discretized according to the number of steps. Then each combination of parameters in these discretized intervals is evaluated on a simulator with multiple rollouts being sampled. After that, the average return is calculated.

For evaluation on the real system, the policies are executed on the real world counterpart. Multiple rollouts are sampled to calculate the mean performance. Since we are checking if a direct transferability is given, no further training will be performed. If a trained policy fails on the real system, another policy trained with different hyperparameters and another random seed will be used. If there is no success, the policy is defined as not transferable to the real system.

State Augmentation in ARPL
There is a special case for ARPL because it adds the domain parameters to the state which the agent perceives. On the real platform the exact domain parameters are not known. This is why ARPL will be evaluated twice in simulation. One time with the standard setting which will forward the parameter configuration ξ to the policy and one time were the fixed nominal domain parameters are fed to the agent. We will name the results as *ARPL FIXED*.

5 Experiments

In order to perform an evaluation of a domain randomization algorithm, policies need to be learned. All of the evaluated algorithms use PPO but have access to the environments in which the trajectories are sampled. These trajectories are then used to perform a policy update using a method of PPO. In the following we will explain the training setup for the different platforms and algorithms.

5.1 Training

For the training process, multiple hyperparameter configurations are evaluated. But since the performance of RL algorithms is sensitive to the hyperparameters [40], hand tuning the parameters might not be the best option. Because it is difficult to predict which parameters have to be randomized, ARPL and ADR are set to randomize all domain parameters.

5.1.1 Furutua Pendulum

On the Furuta pendulum, a feed forward neural network policy with two 64 neuron hidden layers is used. A similar network is used as critic. The activation function on the hidden layers is set to tanh.

PPO
For the training of PPO, the discount factor γ was set to 0.99, the batch size for the critic and policy update was set to 64 and the parameter λ for the GAE was set to 0.95. The learning rate used for the critic and the policy was 0.0005 and 10 update steps were performed in each iteration. The clipping of PPO was set to 0.1 while the initial standard deviation of the exploration strategy was set to 1.0.

UDR
For the training of the subroutine PPO, the randomization space was set according to Table A.5. The parameters of the involved PPO algorithm remained unchanged.

ARPL
For training ARPL, the parameters of the underlying PPO are the same as in the plain setting, but the learning rate is set to 0.0003 for both actor and critic. Adversarial samples were applied to all domain parameters, where the probability of applying adversarials to the domain parameters was 0.33 and the scaling factor was set to 0.05. For process noise the settings were 0.1 for the probability and 0.006 for the scaling factor. Lastly, the adversarial observation noise was applied with a probability of 0.1 and a scale of 0.1.

ADR
In order to train ADR, 10 particles are used to generate trajectories of proposed domain parameters. Each of them is reset after 200 iterations. The particles consist of a feed forward neural network with one 100 neuron hidden layer as an actor and a critic network with two 100 neuron hidden layers. Here, the nonlinearities are set to be tanh. Moreover, changes to parameter values proposed by the particles are clipped to 0.01 times the nominal value and the initial domain configurations are sampled using a uniform distribution with the means set to the nominal parameters and a halfspan of 0.5 times the nominal parameter. Our discriminator is a neural network with 1 layer of 128 neurons and tanh as the nonlinearity in the hidden layers while a sigmoid is used on the output. The discriminator performs updates 10 times per iteration. Further, the other parameters of the underlying PPO remain unchanged. The temperature α of ADR which determines the influence of the particles to each other is set to 10 which encourages joint training.

5.1.2 Ball Balancer

For the Ball Balancer, a neural network policy with two 32 neuron layers is used. The activation function of the hidden layers is set to tanh. The network for the value function is similar but only has one output. The parameters of GAE have been set to $\gamma = 0.9995$ and $\lambda = 0.98$. In each iteration the gradient of the critic network is updated three times using the samples. Therefore, the learning rate is initialized with 0.002 and reduced by a learning rate scheduler over time.

The same settings are used for the actor network updates. Moreover, the standard deviation of the exploration strategy is initialized with 0.9. A batch size of 50 is used for the computation of the update steps.
The other algorithms use the same settings but have different learning rates. All other settings are the same as in the Furuta pendulum setup.

5.2 Evaluation

After training several policies with each algorithm, the best ones are selected for further investigation. Then the candidates are evaluated in simulation using the nominal configuration as well as randomized configurations. The configurations of the randomizer, which is used to sample new configurations in every rollout can be found in Table A.5 for the Furuta pendulum and table A.6 for the ball balancer. In addition, a harder setting with greater standard deviations is evaluated on the Furuta pendulum. It can be found in Table A.3. As a harder setting for the ball balancer, table A.6 changed so the standard deviation or halfspan is a third of the nominal value for every item.
Next, the policies are evaluated using environment grids, where each axis of the two dimensional grid is assigned to a physics parameter. The grid therefore contains every combination of the discrete value along the axes. Each setup is evaluated in simulation for 100 rollouts per configuration. For each algorithm a hand picked policy is evaluated.
For the Furuta pendulum, the following pairs are evaluated in simulation:

- rotary arm length and pendulum link length
- rotary arm mass and pendulum link mass
- motor resistance and gravity
- motor resistance and action delay.

On the ball balancer, the following pairs are evaluated:

- gravity and ball radius
- ball radius and length of the plate.

Finally, the policies trained for the Furuta Pendulum will be evaluated on the real system.

6 Results

After several training runs yielded a variety of policies, the most fit ones had to be found. The best policy of each algorithm was used for evaluation as described in section 5.2.

6.1 Furuta Pendulum

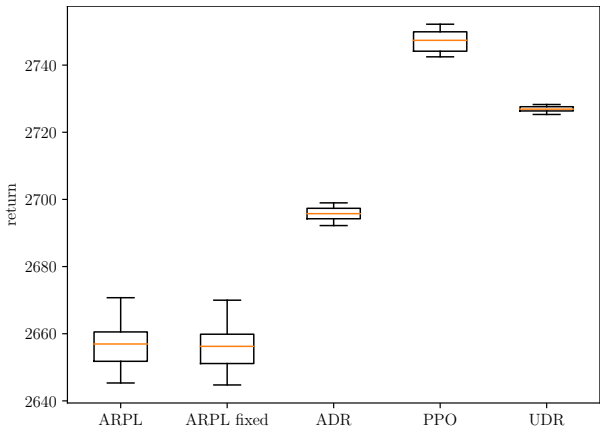
First, the policies were evaluated in simulation using the nominal values. The results are shown in Figure 6.1a and Figure 6.1b. PPO performs best, which is plausible, because the agent has solely experienced the reference environment. It is therefore trained to get the highest reward The success rates based on our metric defined in (4.3) are shown in Figure 6.1b. All agents could solve the task in every rollout. When the simulation was perturbed in every rollout, the PPO agent began to fail more often. The policies which have been learned using sim-2-real methods performed better than PPO which was trained in the nominal settings only. ADR performs slightly better in randomized environments. But in the randomized environments with higher variance, ADR performed significantly better.
The environment grids in Figure 6.2, 6.3, 6.4 and 6.5 show that ADR performs best but in general all methods with domain perturbation are able to learn policies which can succeed in different environment configurations. The grids show the average return over $N = 100$ rollouts, where an average return of approximately 450 is a success. Nevertheless the success rate of the policy learned using ARPL is reduced to zero. This decrease in success can also be seen in the sim-2-real evaluation, where our policies always applied very high voltages and therefore failed right at the beginning of each rollout. We also evaluated the performance under the presence of action delay. As shown in Figure 6.5e, none of the policies is able to perform well under the influence of action delay.

6.2 Ball Balancer

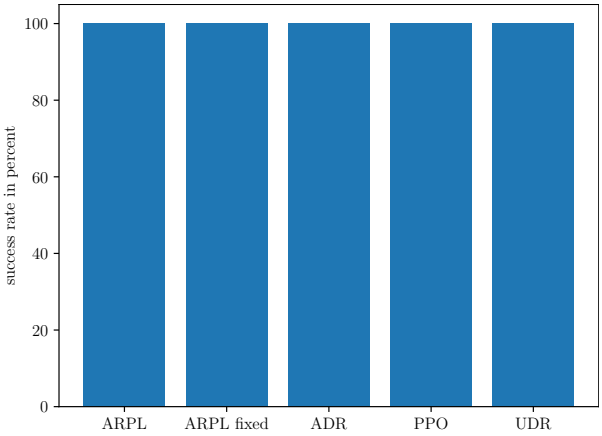
Similar to the Furuta pendulum, the sim-2-sim evaluation shows improvements in the transferability to simulations with different domain parameters, see Figure 6.6. Also, policies trained with ADR and UDR perform best. The environment grids in Figure 6.7 and Figure 6.8 show that even plain PPO performed relatively well on all of the grid environments. In the harder sim-2-sim evaluation with a larger variance in Figure 6.6, PPO did not perform well, while the other methods achieved higher rewards.

6.3 Discussion

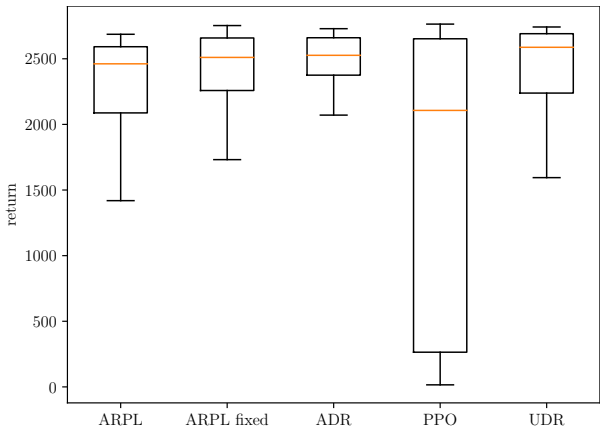
The results from the sim-2-sim evaluation show, that the usage of sim-2-real methods did indeed improve the transferability to environments with different configurations. Yet, the inability to transfer the policies generated using our setup of ARPL to the real system leads to the question if an extensive hyperparameter search is necessary to train better policies. Since DR and especially ARPL change the underlying MDP, there is a chance that the perturbations are too big, making the agent only learn to solve the perturbed MDP. In case of random perturbations the agent is unlikely to overfit but in ARPL the perturbations are partially deterministic but only applied with a certain probability. Further studies could evaluate the impact of the probability to apply adversarial samples and the performance of the trained policies. ADR performed best on both platforms. Due to a good selection of the randomization space, UDR was also able to perform well. It performed best on the real system. It would be very interesting to see how small changes in the domain randomization space affect the performance of the algorithm. However, each parameter which is checked changed, significantly improves the computation time needed to perform the experiments. Therefore, our evaluation is very basic so far.



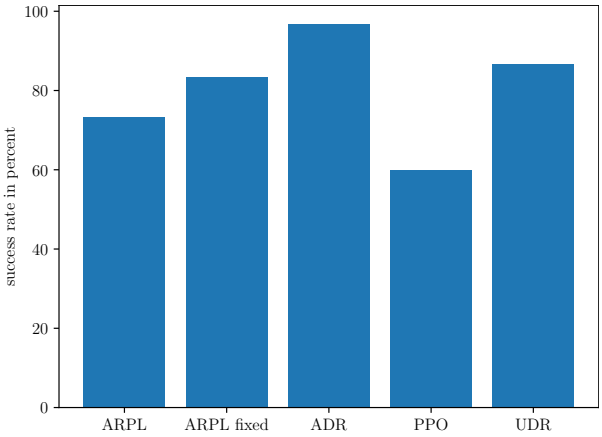
(a) Average returns of policies trained with the different methods. In an environment with the nominal configuration.



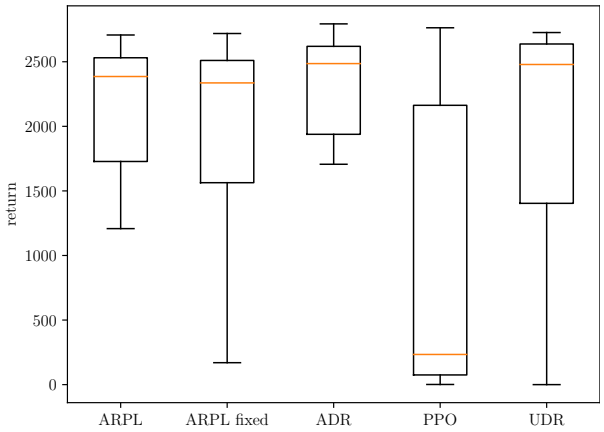
(b) Success rate of policies trained with the different methods. In an environment with the nominal configuration.



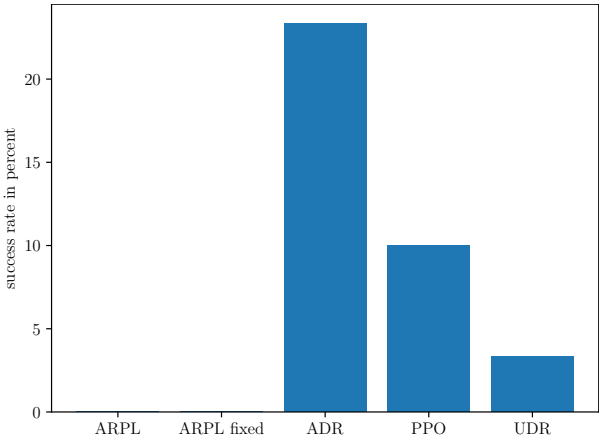
(c) Average returns of policies trained with the different methods. In environments with randomized configurations.



(d) Success rate of policies trained with the different methods. In environments with randomized configurations.



(e) Average returns of policies trained with the different methods. In environments with harder randomized configurations.



(f) Success rate of policies trained using the different methods. In environments with harder randomized configurations.

Figure 6.1.: Results of the sim-to-sim comparison on the Furuta pendulum using N=100 rollouts per policy.

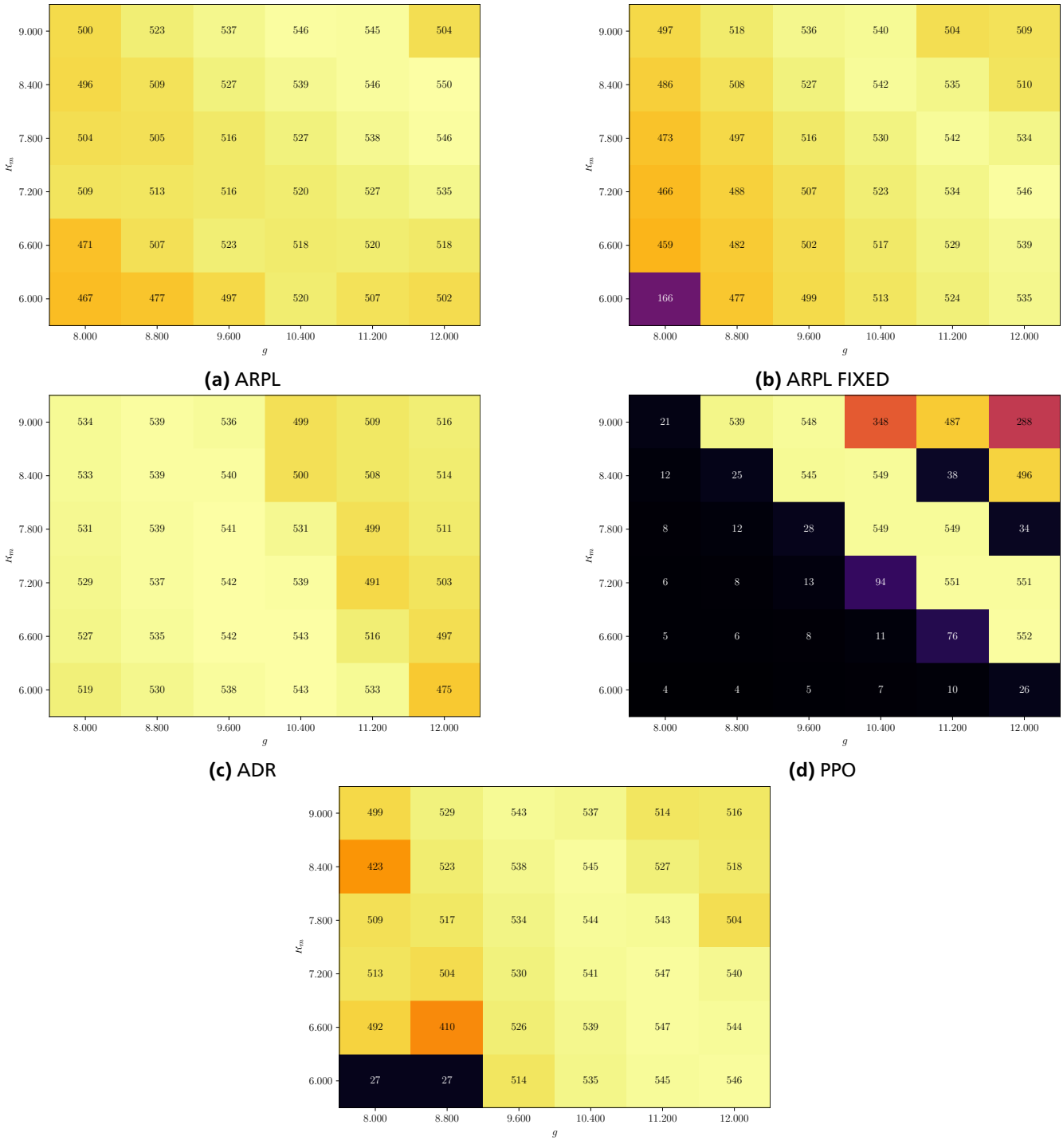
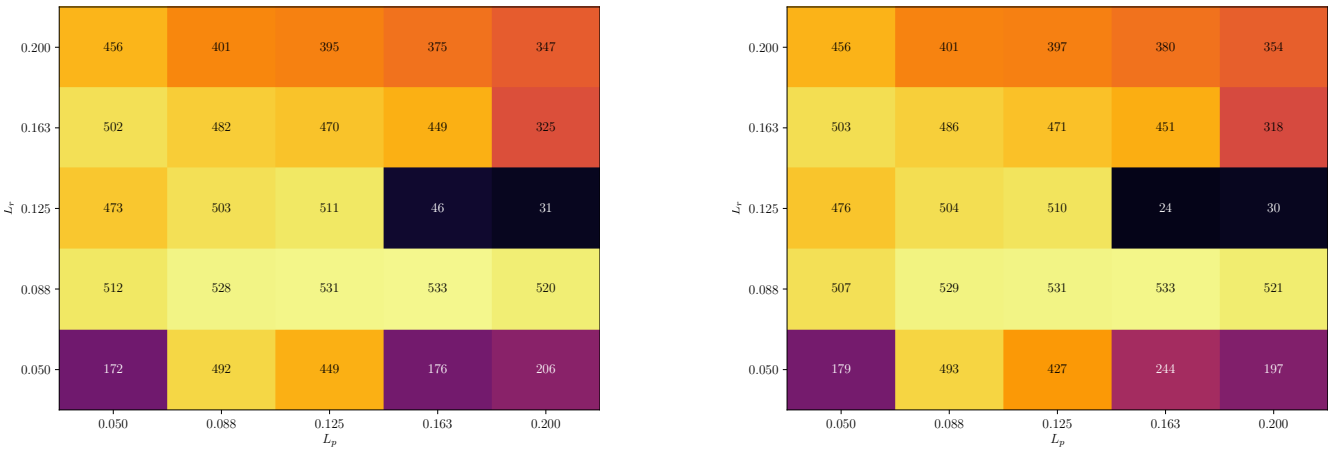
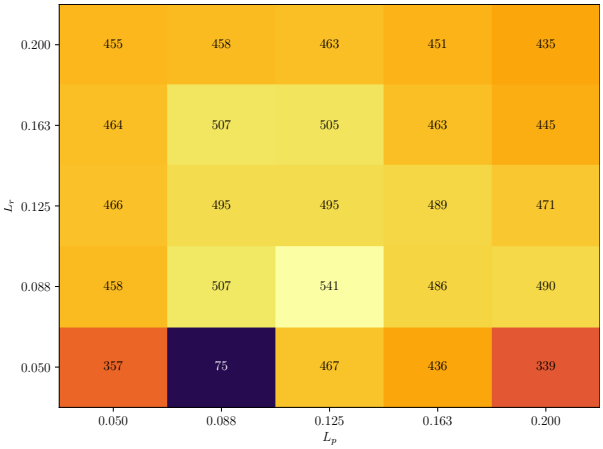


Figure 6.2.: Environment grid of the Furuta pendulum with R_m and g variated.

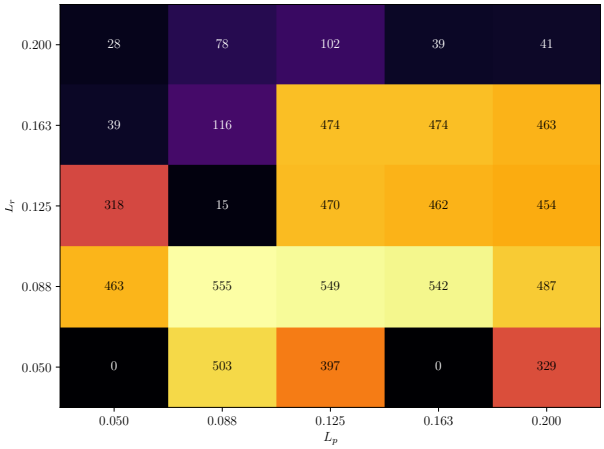


(a) ARPL

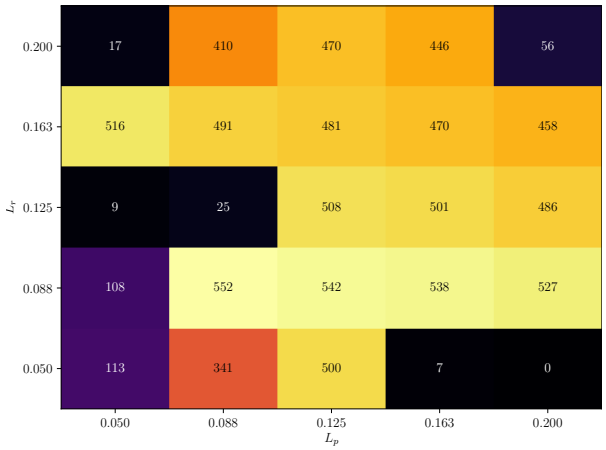
(b) ARPL FIXED



(c) ADR

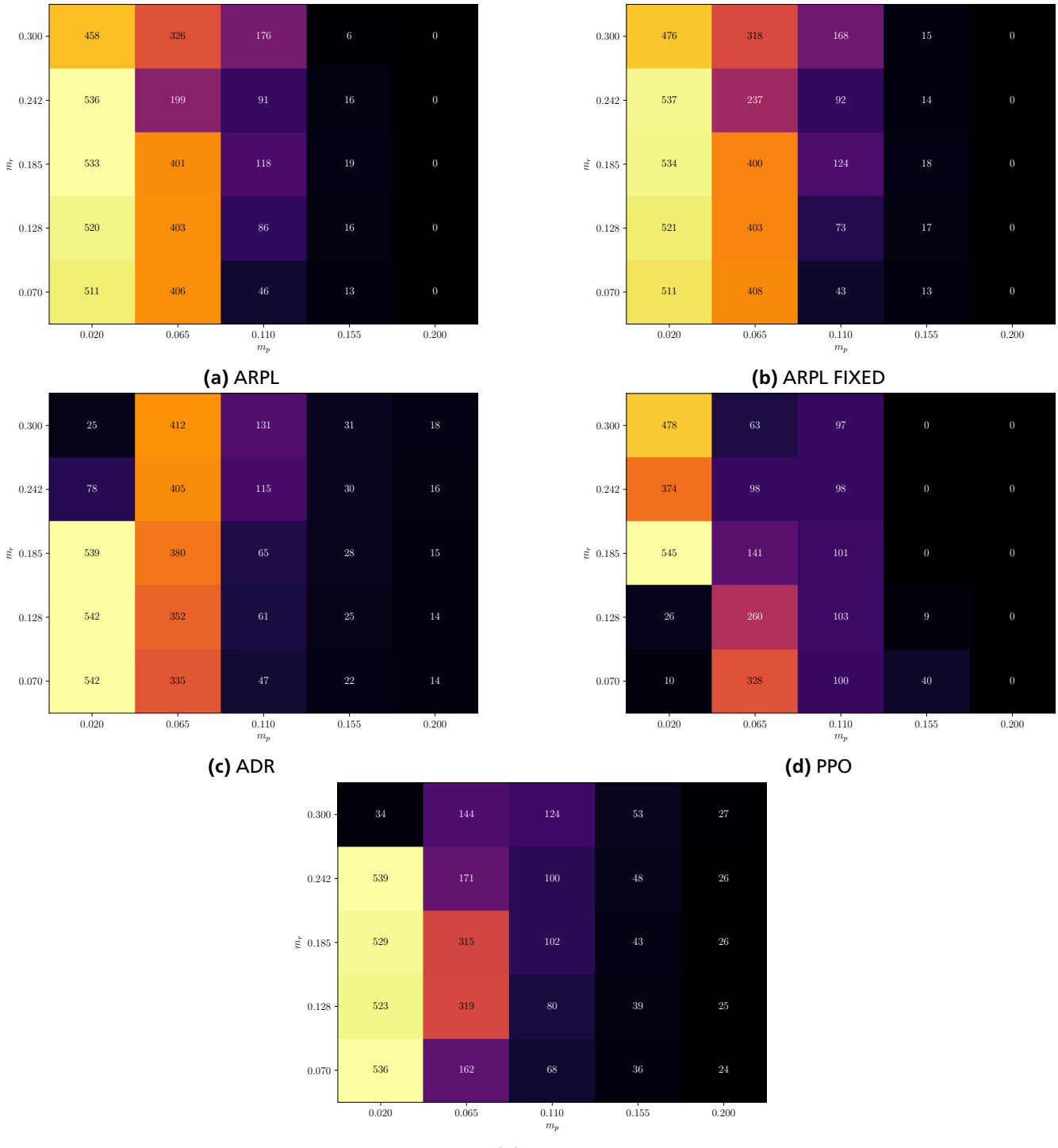


(d) PPO



(e) UDR

Figure 6.3.: Environment grid of the Furuta pendulum with the lengths Lr and Lp variated.



(a) ARPL

(b) ARPL FIXED

(c) ADR

(d) PPO

(e) UDR

Figure 6.4.: Environment grid of the Furuta pendulum with the masses M_r and M_p variated.

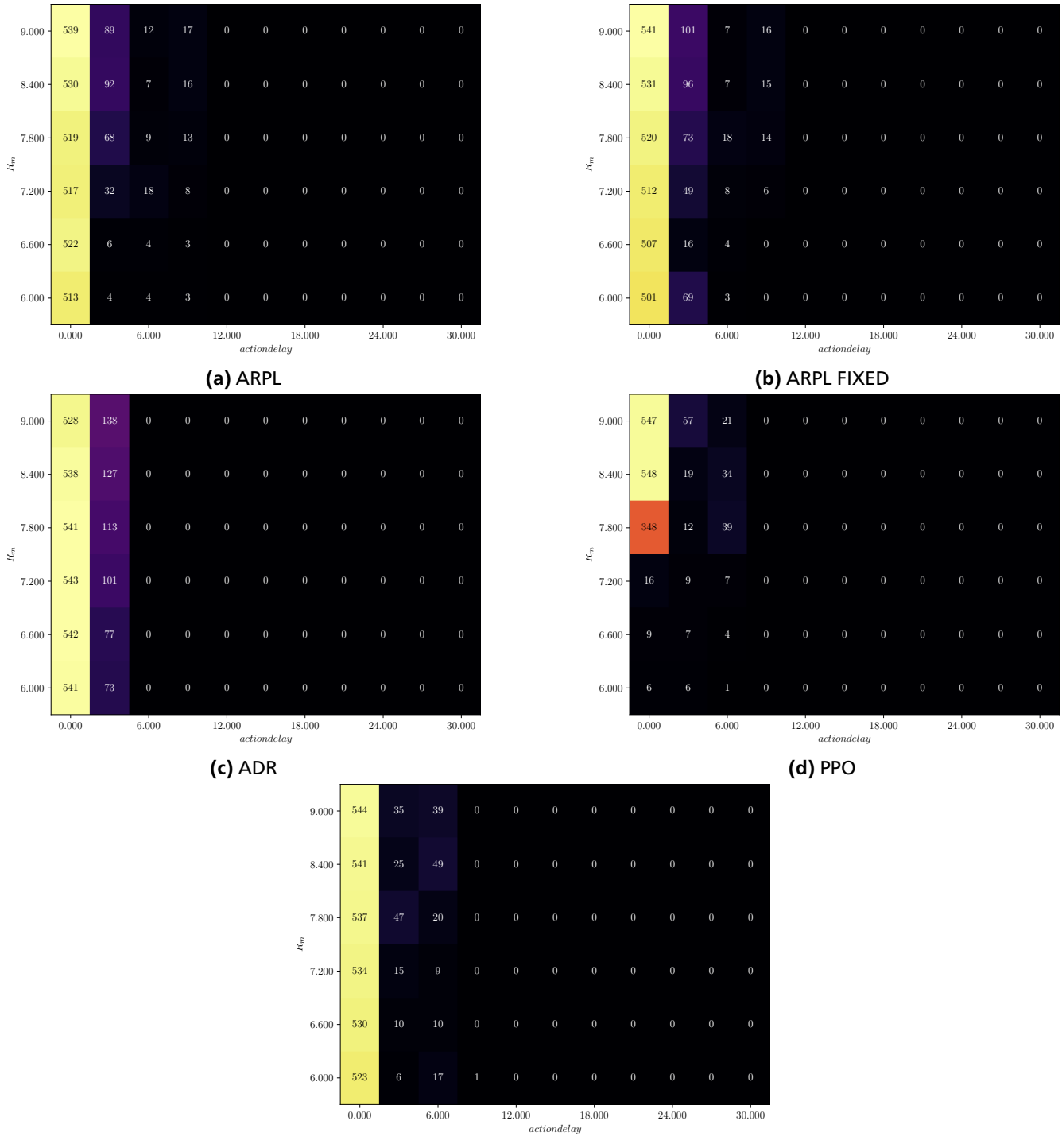
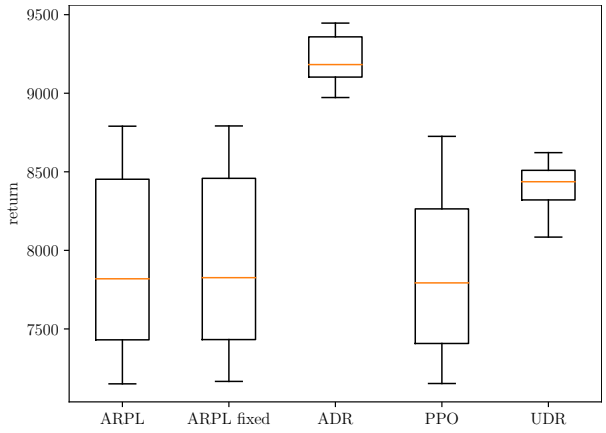
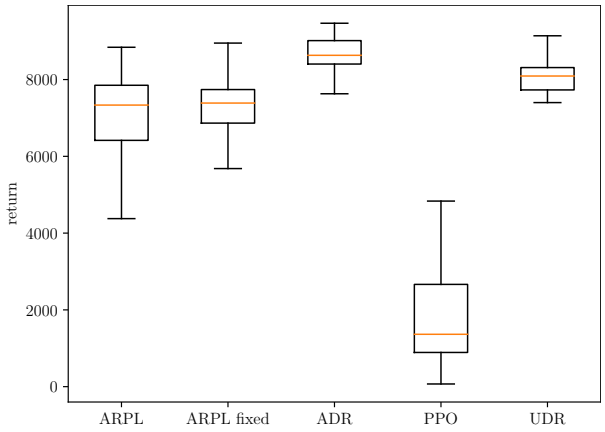


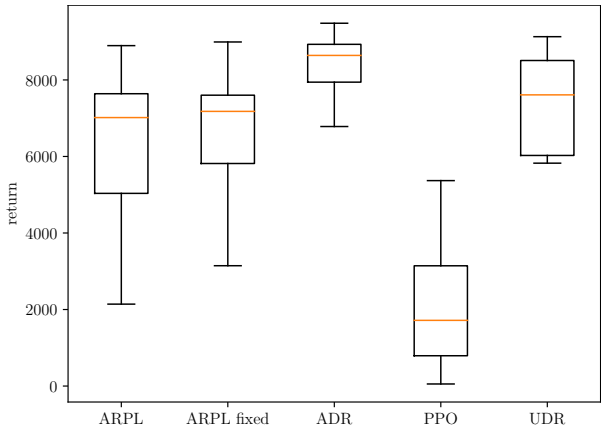
Figure 6.5.: Environment grid of the Furuta pendulum with the motor resistance R_m and action delay variated.



(a) Average returns of policies trained with the different methods. In an environment with the nominal configuration.

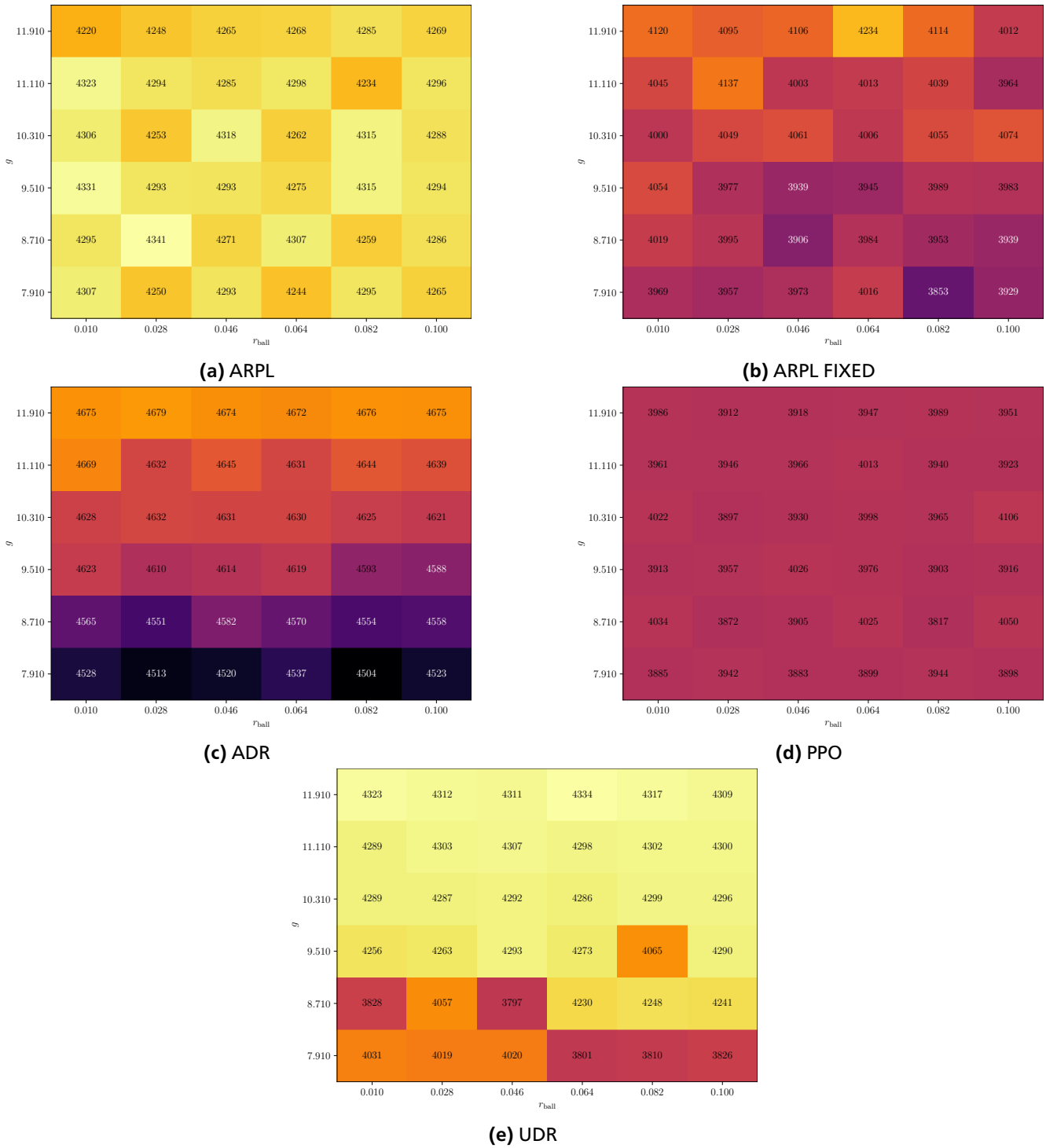


(b) Average returns of policies trained with the different methods. In environments with randomized configurations.



(c) Average returns of policies trained with the different methods. In environments with harder randomized configurations.

Figure 6.6.: Results of the sim-to-sim comparison on the ball balancer using N=100 rollouts per policy.



(a) ARPL

(b) ARPL FIXED

(c) ADR

(d) PPO

(e) UDR

Figure 6.7.: Environment grid of the ball balancer with gravity g and ball radius r_{ball} variated.

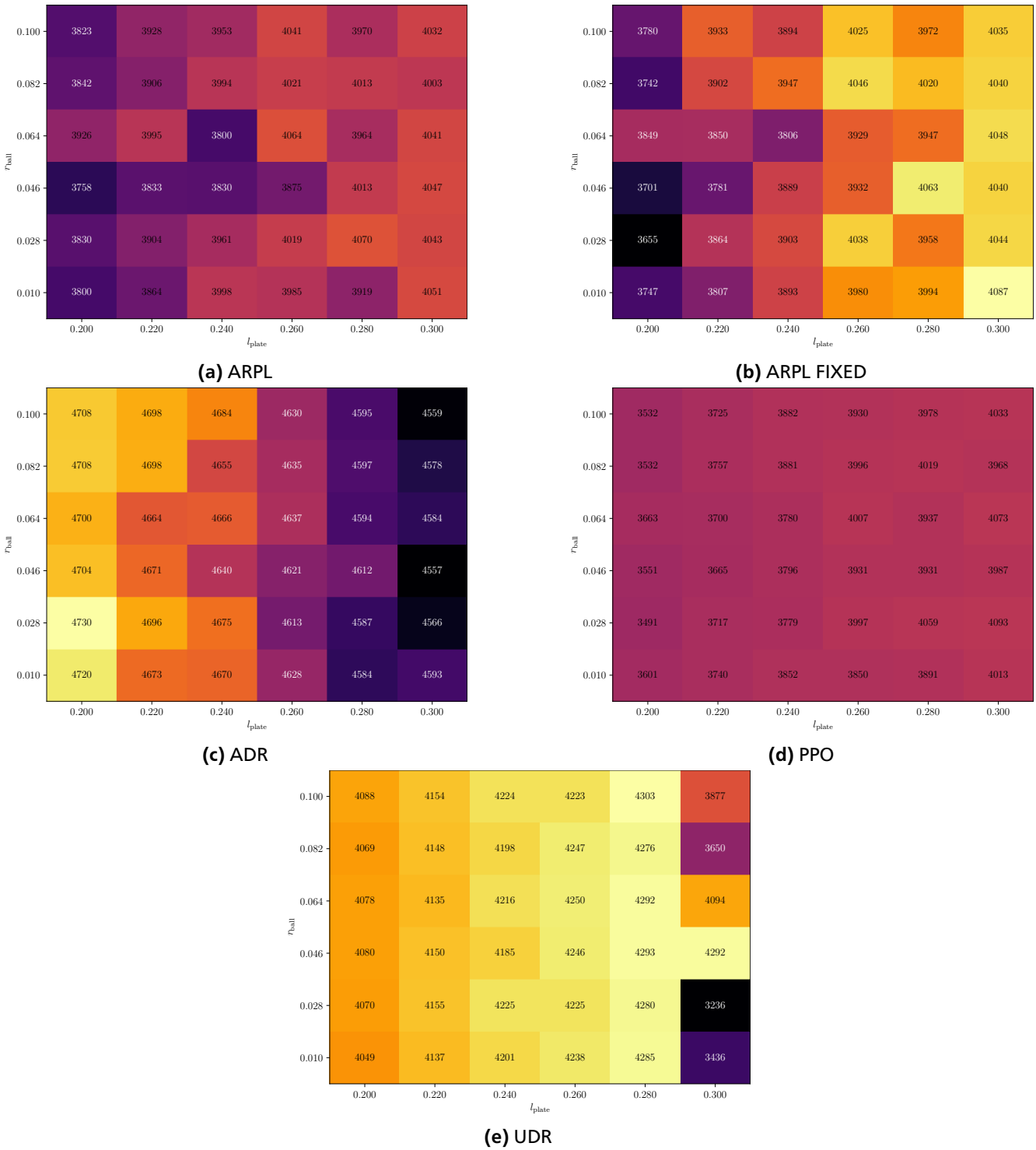


Figure 6.8.: Environment grid of the ball balancer with the ball radius r_{ball} and the plate length l_{plate} variated.

Method	Mean reward	Median reward	Success rate
ARPL	1.67	0.71	0.0%
ARPL FIXED	1.69	0.70	0.0%
ADR	2220.96	2373.19	83.33%
PPO	331.74	330.66	0.0%
UDR	2523.82	2523.98	100%

Table 6.1.: Sim-2-real results on the Furuta Pendulum

7 Conclusion and Outlook

With the rise of RL, there is growing interest in the transfer of behavior from source to target domains. There are multiple approaches on how to achieve this but domain randomization is very promising because system identification cannot rule out innate modeling errors inherent to simple physical models. Since there is an abundance of new methods in the field of domain randomization, but only a few benchmarks, we set out to evaluate the performance of multiple sim-2-real methods when used to randomize the source domain for training PPO. The evaluation showed that policies trained with (plain) PPO were not transferable to the real system and underperformed in sim-2-sim benchmarks. Moreover UDR showed very good results but did not always converge to well suited policies. It performed better in sim-2-real evaluation than in simulation. A reason for this could be the large variance used to evaluate the algorithms in sim-2-real. In our philosophy to only benchmark the best results of each algorithm, the hyperparameters were handcrafted. However, examining the influences of different hyperparameter settings in a systematical way (e.g. [41]) would be a good follow up. Additional work could be the examination of the influence of the domain randomization space. In our experiments ARPL create well transferable policies. A problem could be the size of the neural networks in use, since the input space is augmented by the states. In addition, the policies that used the augmented state of ARPL did not perform significantly worse, when the inputs on the states were fixed during the evaluation. The use of system identification could increase the performance on networks with the physics parameters as an input. Additionally, other architectures of neural networks (e.g. LSTM [42]) can be evaluated. The benchmarks conducted in this thesis only focused on a narrow set of benchmarks. Richer benchmarks can be a good follow up work.

ADR, however, was able to learn policies that outperformed the other methods on both platforms. Yet, ADR looks promising because it is not necessary to fine-tune the domain randomization space.

All in all this thesis provides the foundations of domain randomization in deep reinforcement learning. The benchmark design was very simple and could be enhanced in the future. However, the results of the benchmarks were able to show the importance of sim-2-real approaches. Of all evaluated algorithms, ADR showed the best results.

Bibliography

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[2] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, pp. 1889–1897, 2015.

[3] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *CoRR*, vol. abs/1512.04455, 2015.

[4] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *ECAL*, 1995.

[5] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8973–8979, IEEE, 2019.

[6] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, IEEE, 2017.

[7] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization," *arXiv preprint arXiv:1904.04762*, 2019.

[8] A. Mandlekar, Y. Zhu, A. Garg, L. Fei-Fei, and S. Savarese, "Adversarially robust policy learning: Active construction of physically-plausible perturbations," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3932–3939, IEEE, 2017.

[9] F. Muratore, F. Treede, M. Gienger, and J. Peters, "Domain randomization for simulation-based policy optimization with transferability assessment," in *Conference on Robot Learning (CoRL)*, 2018.

[10] A. Rajeswaran, S. Ghotra, S. Levine, and B. Ravindran, "Epopt: Learning robust neural network policies using model ensembles," *CoRR*, vol. abs/1610.01283, 2016.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[13] S. Koos, J. Mouret, and S. Doncieux, "The transferability approach: Crossing the reality gap in evolutionary robotics," *IEEE Transactions on Evolutionary Computation*, vol. 17, pp. 122–145, Feb 2013.

[14] J. C. Zagal, J. Ruiz-del Solar, and P. Vallejos, "Back to reality: Crossing the reality gap in evolutionary robotics," in *IAV 2004 the 5th IFAC Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal*, 2004.

[15] J.-C. Zufferey, A. Guanella, A. Beyeler, and D. Floreano, "Flying over the reality gap: From simulated to real indoor airships," *Autonomous Robots*, vol. 21, no. 3, pp. 243–254, 2006.

[16] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. W. Pachocki, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *CoRR*, vol. abs/1808.00177, 2018.

[17] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–8, IEEE, 2018.

[18] L. Ljung, "System identification," *Wiley Encyclopedia of Electrical and Electronics Engineering*, 2001.

[19] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, and S. Birchfield, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 969–977, 2018.

[20] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, pp. 1329–1338, 2016.

[21] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, "Benchmarking reinforcement learning algorithms on real-world robots," *arXiv preprint arXiv:1809.07731*, 2018.

[22] N. A. Lynnerup, L. Nolling, R. Hasle, and J. Hallam, "A survey on reproducibility by evaluating deep reinforcement learning algorithms on real-world robots," *arXiv preprint arXiv:1909.03772*, 2019.

[23] T. Haarnoja, *Acquiring Diverse Robot Skills via Maximum Entropy Deep Reinforcement Learning*. PhD thesis, UC Berkeley, 2018.

[24] K. Lowrey, S. Kolev, J. Dao, A. Rajeswaran, and E. Todorov, "Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system," in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pp. 35–42, IEEE, 2018.

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[26] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[27] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[28] M. P. Deisenroth, G. Neumann, J. Peters, *et al.*, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[29] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, pp. 317–328, Springer, 2005.

[30] F. Gomez and J. Schmidhuber, "Evolving modular fast-weight networks for control," in *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005* (W. Duch, J. Kacprzyk, E. Oja, and S. Zadrożny, eds.), (Berlin, Heidelberg), pp. 383–389, Springer Berlin Heidelberg, 2005.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[32] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2015.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[34] Y. Liu, P. Ramachandran, Q. Liu, and J. Peng, "Stein variational policy gradient," 2017.

[35] Q. Liu and D. Wang, "Stein variational gradient descent: A general purpose bayesian inference algorithm," in *Advances In Neural Information Processing Systems*, pp. 2378–2386, 2016.

[36] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. D. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," *CoRR*, vol. abs/1810.05687, 2018.

[37] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[38] V. Gullapalli, J. A. Franklin, and H. Benbrahim, "Acquiring robot skills via reinforcement learning," *IEEE Control Systems Magazine*, vol. 14, pp. 13–24, Feb 1994.

[39] K. Furuta, M. Yamakita, and S. Kobayashi, "Swing-up control of inverted pendulum using pseudo-state feedback," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 206, pp. 263–269, 2019/09/17 1992.

[40] H. Mania, A. Guy, and B. Recht, “Simple random search provides a competitive approach to reinforcement learning,” *CoRR*, vol. abs/1803.07055, 2018.

[41] H. Hoos, “An efficient approach for assessing hyperparameter importance,”

[42] B. Bakker, “Reinforcement learning with long short-term memory,” in *Advances in neural information processing systems*, pp. 1475–1482, 2002.

A Appendix

A.1 Domain Parameters

Symbol	Value	Description	Unit
g	9.81	gravity constant	$\mathrm{m\,s^{-2}}$
m_{ball}	0.003	mass of the ball	kg
r_{ball}	0.019 625	radius of the ball	m
l_{plate}	0.275	length of the (square) plate	m
r_{arm}	0.0254	distance between the servo output gear shaft and the coupled joint	m
K_g	70	gear ratio	—
η_g	0.9	gearbox efficiency	—
J_l	5.2822×10^{-5}	load moment of inertia	$\mathrm{kg\,m^2}$
J_m	4.6063×10^{-7}	motor moment of inertia	$\mathrm{kg\,m^2}$
k_m	0.0077	motor torque constant	$\mathrm{N\,m\,A^{-1}}$
R_m	2.6	motor armature resistance	—
η_m	0.69	motor efficiency	—
B_{eq}	0.015	equivalent viscous damping coefficient w.r.t. load	$\mathrm{N\,m\,s\,rad^{-1}}$
c_{frict}	0.05	viscous friction coefficient	$\mathrm{N\,s\,m^{-1}}$
U_{xpos}	0.28	voltage required to move the x servo in positive dir	V
U_{xneg}	−0.10	voltage required to move the x servo in negative dir	V
U_{ypos}	0.28	voltage required to move the y servo in positive dir	V
U_{yneg}	−0.074	voltage required to move the y servo in negative dir	V
offset_x	0	offset of the motor shaft on the x axis	rad
offset_y	0	offset of the motor shaft on the y axis	rad

Table A.1.: The nominal physics parameters of the Quanser ball balancer.

Symbol	Value	Description	Unit
g	9.81	gravity	$\mathrm{m\,s^{-2}}$
R_m	8.4	motor resistance	Ω
k_m	0.042	motor back-emf constant	$\mathrm{V\,s\,rad^{-1}}$
m_r	0.095	rotary arm mass	kg
L_r	0.085	rotary arm length	m
D_r	5×10^{-6}	rotary arm viscous damping	$\mathrm{N\,m\,s\,rad^{-1}}$
m_p	0.024	pendulum link mass	kg
L_p	0.129	pendulum link length	m
D_p	1×10^{-6}	pendulum link viscous damping	$\mathrm{N\,m\,s\,rad^{-1}}$

Table A.2.: The nominal physics parameters of the Quanser Qube.

A.2 Randomization settings

Symbol	Type (Uniform / Normal)	Mean	Halfspan (Uniform) / Standard Deviation (Normal)	Lower Bound
R_m	Normal	R_m	$R_m/3$	0.0001
M_p	Normal	M_p	$M_p/3$	0.0001
M_r	Normal	M_r	$M_r/3$	0.0001
L_p	Normal	L_p	$L_p/3$	0.0001
L_r	Normal	L_r	$L_r/3$	0.0001
g	Normal	g	$g/300$	0.0001

Table A.3.: A hard randomization setting for the Furuta Pendulum. In the mean and halfspan, standard deviation columns, the symbol means the nominal value.

Symbol	Type (Uniform / Normal)	Mean	Halfspan (Uniform) / Standard Deviation (Normal)	Lower Bound
R_m	Uniform	R_m	$R_m/5$	0.001
M_p	Uniform	M_p	$M_p/5$	0.0001
M_r	Uniform	M_r	$M_r/5$	0.0001
L_p	Uniform	L_p	$L_p/5$	0.0001
L_r	Uniform	L_r	$L_r/5$	0.0001
g	Uniform	g	$g/5$	0.001
k_m	Uniform	k_m	$k_m/5$	0.0001
D_r	Uniform	D_r	$D_r/5$	1e−9
D_p	Uniform	D_p	$D_p/5$	1e−9

Table A.4.: The randomization setting used for UDR on the Furuta Pendulum. In the mean and halfspan, standard deviation columns, the symbol means the nominal value.

Symbol	Type (Uniform / Normal)	Mean	Halfspan (Uniform) / Standard Deviation (Normal)	Lower Bound
R_m	Normal	R_m	$R_m/5$	0.001
M_p	Normal	M_p	$M_p/5$	0.0001
M_r	Normal	M_r	$M_r/5$	0.0001
L_p	Normal	L_p	$L_p/5$	0.0001
L_r	Normal	L_r	$L_r/5$	0.0001
g	Normal	g	$g/5$	0.001
k_m	Normal	k_m	$k_m/5$	0.0001
D_r	Normal	D_r	$D_r/5$	1e−9
D_p	Normal	D_p	$D_p/5$	1e−9

Table A.5.: A randomization setting for the Furuta Pendulum. In the mean and halfspan, standard deviation columns, the symbol means the nominal value.

Symbol	Type (Uniform / Normal)	Mean	Halfspan (Uniform) / Standard Deviation (Normal)	Lower Bound
g	Normal	g	$g/5$	1e-4
m_{ball}	Normal	m_{ball}	$m_{ball}/5$	1e-4
r_{ball}	Normal	r_{ball}	$r_{ball}/5$	1e-3
l_{plate}	Normal	l_{plate}	$l_{plate}/5$	1e-2
r_{arm}	Normal	r_{arm}	$r_{arm}/5$	1e-4
K_g	Normal	K_g	$K_g/4$	1e-2
J_l	Normal	J_l	$J_l/4$	1e-6
J_m	Normal	J_m	$J_m/4$	1e-9
k_m	Normal	k_m	$k_m/4$	1e-4
R_m	Normal	R_m	$R_m/4$	1e-4
η_g	Uniform	η_g	$\eta_g/4$	1e-4
η_m	Uniform	η_m	$\eta_m/4$	1e-4
B_{eq}	Uniform	B_{eq}	$B_{eq}/4$	1e-4
c_{frict}	Uniform	c_{frict}	$c_{frict}/4$	1e-4
U_{xpos}	Uniform	U_{xpos}	$U_{xpos}/3$	
U_{xneg}	Uniform	U_{xneg}	$U_{xneg}/3$	
U_{ypos}	Uniform	U_{ypos}	$U_{ypos}/3$	
U_{yneg}	Uniform	U_{yneg}	$U_{yneg}/3$	
offset_x	Uniform	$offset_x$	$6\pi/180$	
offset_y	Uniform	$offset_y$	$6\pi/180$	

Table A.6.: A randomization setting for the Ball Balancer. In the mean and halfspan, standard deviation columns, the symbol means the nominal value.